Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Deep Reinforcement Learning Applied to Byzantine Agreement

Semester Thesis

Janka Möller

jamoelle@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Darya Melnyk, Oliver Richter
Prof. Dr. Roger Wattenhofer
Prof. Dr. Renato Renner

February 24, 2020

Acknowledgements

I thank Darya Melnyk and Oliver Richter for the fruitful discussions and food for thought in the weekly meetings, as well as their support throughout the project. I thank Prof. Dr. Roger Wattenhofer and Prof. Dr. Renato Renner for their supervision and support.

Abstract

We use Deep Reinforcement Learning to simulate worst-case behaviour of failing nodes for reaching Byzantine Agreement. In particular we study the King algorithm for the synchronous communication model and the Ben-Or algorithm for the asynchronous communication model. We find that the byzantine agent successfully learns strategies leading to maximal run time in various settings of both algorithms and we study the resulting actions in detail. We conclude that Deep Reinforcement Learning is a useful tool to simulate worst-case behaviour in the Byzantine Agreement Problem.

Contents

Chapter 1

Introduction

In distributed computing systems, the most dangerous event is not when a node fails and sends no signal, but when a failing node exhibits arbitrary behaviour. The approach for solving byzantine agreement is to develop algorithms to ensure that the non-failing nodes reach consensus, despite a certain maximal number of failing nodes. In the case of an asynchronous non-cryptographic model, there are currently only few algorithms that reach byzantine agreement, they either have a long runtime or tolerate only few failing nodes. [1]

The simulation of the behaviour of the failing nodes can be insightful to study byzantine agreement further and develop efficient algorithms.
Our aim is to model worst-case behaviour of the failing nodes, in terms of achieving maximal run time of a given byzantine agreement algorithm, using Deep Reinforcement Learning. In particular, we consider the King algorithm [2] for the synchronous model and the Ben-Or algorithm [3] in the asynchronous setting. To incentivise the failing nodes to exhibit worst-case behaviour, we formulate the byzantine agreement algorithms as a reinforcement learning environment (see Chapter 4) and reward the failing nodes for every round in which the correct nodes have not reached consensus. We show that the trained byzantine agents (i.e. the failing nodes) achieve maximal run time of both the King and the Ben-Or algorithm respectively. We study various modifications of the algorithms, including one where the agent has to determine the initial values, which she successfully learns.

Our experiments show that Deep Reinforcement Learning can be used as a tool for simulating worst-case byzantine behaviour. Deep Reinforcement Learning (DRL) is a model-free machine learning technique combining classical reinforcement learning with neural networks [4]. DRL has gathered attention when the learners achieved super-human performance levels in some Atari 2600 games [4].

First, we give an overview of related work in Chapter 2. We introduce the theoretical framework of byzantine agreement and the algorithms we study in Chapter 3. Chapter 4 introduces classical and deep reinforcement learning with a focus on Q-learning. We describe the experimental setup and our implementation in Chapter 5, present our results in Chapter 6 and discuss them in Chapter 7.

Chapter 2

Related Work

Researchers have recently applied Deep Reinforcement Learning to graph problems in [5] and [6]. In [5], the authors used DRL with Graph Convolutioal Networks for channel allocation in WLANs. [6] apply Graph Neural Networks in combination with DRL to a routing optimization problem. In contrast, our work focuses more specifically on faults of nodes. Q-learning was applied to fault tolerant control in [7] and to fault handling in self-organizing system by [8]. We are not directly interested in fault handling in this work, but in the simulation of byzantine nodes through DRL. An example for the use of DRL for simulations is [9] where the authors simulate crowd navigation. More closely related to our work is the simulation of worst-case adversarial behaviour on blockchain by [10], where the researchers use DRL to identify attack strategies on incentive protocols such as selfish-mining.
Byzantine faults in relation to machine learning has been focused on studying distributed machine learning computations and their robustness against byzantine attacks [11, 12, 13, 14].
To the best of our knowledge, we are the first to use DRL to simulate worst-case byzantine behaviour with respect to the King and the Ben-Or algorithm.

Chapter 3

Byzantine Agreement

We consider a system of nodes, where any two nodes can communicate directly. In the context of byzantine agreement we are interested in the case where some nodes in the system are failing, i.e. they exhibit arbitrary behaviour including sending different signals to different nodes. In the following, we will refer to the failing nodes as *byzantine nodes* and to the non-failing nodes as *correct nodes*. Furthermore, we use n to denote the total number of nodes in the system, f for the number of byzantine nodes, which leaves us with $n - f$ correct nodes. We assume that byzantine nodes can not forge their sender address and therefore can not impersonate other nodes. However, the byzantine nodes can be controlled by a single agent - which will be crucial in Chapter 5. We say byzantine agreement is reach, if we have consensus in the described system. [1]

3.1 Consensus

Definition 3.1 (Byzantine Agreement)**.** There are n nodes, of which at most f might be byzantine, i.e. at least $n-f$ nodes are correct. Every node i starts with input value v_i. The correct nodes must decide for one of those values, satisfying the following properties: [15]

- **Agreement** All correct nodes decide for the same value.
- **Termination** All correct nodes terminate in a finite number of steps.
- **Validity** The decision value must be the input value of a node.

There are various forms of validity. For our algorithms we will focus on the weakest definition of validity, called *all-same validity*.

Definition 3.2 (All-Same Validity)**.** If **all** correct nodes start with the input value v, the decision value must be v. [1]

To study byzantine agreement, we need to keep in mind the following theorem:

Theorem 3.3. *A network with n nodes cannot reach byzantine agreement with $f \geq n/3$ byzantine nodes. [16]*

Proof. Essentially one can show that a system with $f \geq n/3$ byzantine nodes can not satisfy both all-same validity and agreement by a proof of contradiction. We give a more detailed proof in the appendix. □

Definition 3.4 (Synchronous and Asynchronous Model)**.** In the **synchronous model** all nodes operate in synchronous rounds. Where one round consists of sending and receiving messages and doing some local computation. In the **asynchronous model** each node only awaits a certain number of messages of the current round. [1]

3.2 The Synchronous Model

Algorithm 1 King Algorithm for $f < n/3$ (from [1])

```
1:  x= my input value
2:  for phase =1 to f+1 do
    Round 1
3:      Broadcast value(x)
    Round 2
4:      if some value(y) received at least n − f times then
5:          Broadcast propose(y)
6:      end if
7:      if some propose(z) received more than f times then
8:          x = z
9:      end if
    Round 3
10:     Let node v_i be the predefined king of this phase i
11:     The king v_i broadcasts its current value w
12:     if received strictly less than n − f propose(y) then
13:         x = w
14:     end if
15: end for
```

We give the pseudo-code for the King algorithm in Algorithm 1.

There are a few important statements to be made about the King algorithm.

Lemma 3.5. *The King algorithm satisfies all-same validity. [1]*

Proof. If all correct nodes start with value v, every correct node will fulfil Line 4 and propose v. Then no correct node fulfils Line 7 and will therefore not change

its value in Line 8. Neither will any correct node change to the kings value in Line 13, as no correct node fulfils Line 12. This holds for all following phases. [1] □

Lemma 3.6. *There is always at least one phase with a correct king. And after a phase with a correct king, the correct nodes will not change their values v anymore. [1]*

Proof. The first part is trivial, since we have $f+1$ phases, f byzantine nodes and a different (predefined) king in each phase, one of the kings must be a correct node. The proof of the second part is given in the appendix. [1] □

3.3 The Asynchronous Model

Algorithm 2 Ben-Or for $f < n/10$ (from [1])

```
1: x_u ∈ {0,1}                                              ▷ input bit
2: r=1                                                      ▷ round
3: Broadcast propose(x_u,r)
4: repeat
5:     Wait until n − f propose messages of current round r arrived
6:     if at least ⌊n/2⌋ + 3f + 1 propose messages contain same value x then
7:         x_u = x, decided = true
8:     else if at least ⌊n/2⌋ + f + 1 propose messages contain same value x
   then
9:         x_u = x
10:    else
11:        choose x_u randomly, with Pr[x_u = 0] = Pr[x_u = 1] = 1/2
12:    end if
13:    r = r+1
14:    Broadcast propose(x_u,r)
15: until decided (see Line 7)
16: decision = x_u
```

The Ben-Or algorithm is given in Algorithm 2. Note, that not both x and y where $x \neq y$ can be chosen in Line 9. For the sake of contradiction assume both x and y with $x \neq y$ implying both were proposed by at least $\lfloor n/2 \rfloor + 1$ correct nodes, which is impossible since $2 \cdot (n/2 + 1) = n + 2 > n - f$. [1]

One can show that it solves byzantine agreement: [1]

- **All-Same Validity** If every correct node has the same initial value v, every node will terminate in the first phase since $n - 2f \geq \lfloor n/2 \rfloor + 3f + 1$ for $f < n/10$.

- **Agreement** For two nodes at most $2f$ messages can be different. If a node terminates to value v_{final} (Line 7) it has received v_{final} at least $\lfloor n/2 \rfloor + 3f + 1$ times, then all other nodes must have received v_{final} at least $\lfloor n/2 \rfloor + 3f + 1 - 2f = \lfloor n/2 \rfloor + f + 1$ times and fulfil (at least) Line 8.
- **Termination** For termination we essentially need to wait until all correct nodes randomly choose the same value which happens with a probability $2^{-(n-f)+1}$, leading to exponential runtime. Of course, if all correct nodes start with the same initial value the algorithm terminates immediately by all-same validity.

Chapter 4

(Deep) Reinforcement Learning

4.1 Reinforcement Learning

In this section we follow [17] in describing the reinforcement learning problem.

Consider an agent interacting with an environment. Assume that, the agent observes states $s_t \in \mathcal{S}$ then takes an action $a_t \in \mathcal{A}$ and receives a reward $r_t \in \mathbb{R}$. For our purposes we consider discrete time steps t = 0,1,2,3,... The agent acts according to her policy $\pi(a|s_t)$ which is a conditional probability distribution over the actions given the current state s_t. Her goal is to maximise the *expected (discounted) reward* given by $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_{k+1}$, with $0 \leq \gamma \leq 1$, where γ is the discount factor. We assume the states to fulfil the *Markov Property* meaning the response at $t+1$ only depends on the action and state at time t and the reinforcement learning task is a *(finite) Markov Decision Process (MPD)*. This means the process is fully described by the one-step dynamics, i.e. the *transition probabilities*

$$\mathcal{P}^a_{ss'} = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$$

and the expected next reward

$$\mathcal{R}^a_{ss'} = \mathbb{E}(r_{t+1} | s_t = s, a_t = a, s_{t+1} = s')$$

We can now define the *action-value function for policy* π as

$$Q^\pi(s, a) = \mathbb{E}_\pi(R_t | s_t = s, a_t = a)$$

We are interested in the optimal action-value function $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ which satisfies the *Bellman Equation*:

$$Q^*(a, s) = \mathbb{E}(r_{t+1} + \gamma \max_{a'} Q^*(s, a') | s_t = s, a_t = a)$$

There are several methods to solve the reinforcement learning task, such as state-value or policy iteration. In this work we will focus on *Q-learning*.

4.1.1 Q-learning

The above Bellman Equation for the Q-function can be solved using an iterative process, which is shown to converge to the optimal value in [18]. We consider an agent in stage k observing s_k, taking action a_k, receiving reward r_k and observing next state s_{k+1}. Then it should update its action-value function according to:

$$Q_k(s,a) = \begin{cases} (1-\alpha_k)Q_{k-1}(s,a) + \alpha_k[r_k + \gamma \max_{a'} Q_{k-1}(s_{k+1},a')] & \text{if } a = a_k, s = s_k \\ Q_{k-1}(s,a) & \text{otherwise} \end{cases}$$

where α_k is a predefined learning rate. [18]

4.2 Deep Reinforcement Learning

In Deep Reinforcement Learning a neural network is used to approximate the Q-function [4] i.e. $Q(s,a;\theta) \approx Q^*(s,a)$. To find the parameters one can use the loss function

$$L_k(\theta_k) = \mathbb{E}_{s,a}[(y_k - Q(s,a;\theta_k))^2]$$

with $y_k = \mathbb{E}_{s,a}[r + \gamma \max_{a'} Q(s',a';\theta_{k-1})|s,a]$. [4]

In our work, we use the DQN algorithm, which is based on Q-learning. Note, that there exist various other Deep Reinforcement Learning approaches, also ones not using Q-learning [19]

Chapter 5

Implementation

We are ultimately interested in whether a byzantine agent, trained by deep reinforcement learning, can achieve maximal run time of the King or the Ben-Or algorithm. We make use of the fact, that all byzantine nodes can be controlled by one adversary and therefore train a single agent controlling all byzantine nodes simultaneously, where the maximal number of byzantine nodes is given by each algorithm respectively. We use the DQN algorithm by Stable Baselines [20], where we use a 2 hidden layer Multi Layer Perceptron (MLP) network with 64 nodes in each hidden layer. The DQN algorithm uses an ϵ-*greedy* strategy, meaning a random action is chosen with probability ϵ and otherwise the action maximising the current Q-function. The DQN algorithm starts with $\epsilon = 1$ and anneals it to $\epsilon = 0.02$ over the *exploration fraction* of the entire training period, then keeps $\epsilon = 0.02$ constant for the rest of the training. For the environment we follow the formalism of the Gym-package [21]. Essentially the environment is a class which needs to fulfil the following:

- **action_space**, **observation_space** attributes defined as Gym-spaces
- **reset** function to set the beginning of a new game
- **step** function taking the argument *action* and returning the tuple *observation, reward, done, info* where *done* is a bool indicating whether the game has terminated

In our case, the environment is given by either the King or Ben-Or algorithm, which we will present in more detail in the next sections.

In the following we will refer to a full run of a byzantine agreement algorithm as a *game* or an *episode*, and to one time step as *step*. We will refer to the adversary controlling all byzantine nodes as *agent* or *byzantine agent*. We train a separate agent for each of the following versions.

5.1 King Algorithm

The most important aspect to note about the King algorithm is, that each *phase* consists of 3 *rounds* (see Algorithm 1). Therefore, a step of the game corresponds to a round in the algorithm, where we let the agent observe in which round she currently is. The start and end of each round is slightly different from the ones defined in Algorithm 1. The decisive point of when a round has to end is given by when the agent has to take an action, which is in Lines 3, 5, 13 of Algorithm 1.

We have to define who will be king in which phase, where no one can be king more than once and hence one of the kings is a correct node. As shown in Section 3.2, all correct nodes decide after a correct node was king (and stay decided). For the game to be interesting, we therefore let each byzantine node be king once and a correct node be king in the last phase. Furthermore, we need to define the initial values of the correct nodes at the beginning of each episode. We choose them randomly but exclude cases where all correct nodes have the same initial conditions, since in that case no correct node would ever change its value and the byzantine agent had no chance.

5.1.1 Actions and Observations

The messages the nodes exchange are binary values $v \in \{0, 1\}$. An observation consists of a concatenation of:

- vector of values sent by correct nodes corresponding to Lines 3, 5 in Algorithm 1 and a vector of current values of the correct nodes before the king action
- current round of the game $\in \{1, 2, 3\}$ (encoded as a one-hot vector)
- who of the byzantine nodes is king in the current phase (encoded as a one-hot vector)

We make the assumption, that the byzantine agent first receives all messages sent by the correct nodes, then sends her message(s) and the correct nodes finally receive all messages at once. The agent can send an arbitrary value $\in \{0, 1\}$ or send nothing at all for each byzantine node individually, however the messages she sends are the same for all correct nodes. This is a restriction which technically makes the game harder for the agent, but it allows for a much smaller action space and makes the exploration easier. The order of the byzantine agents does not matter, which allows us to reduce the size of the action space further. Hence the actions of the agent are binary vectors of length f containing the values v send by the agent, after the first and the second round and a single number after the third round.

5.1.2 Rewards and Termination

We check whether all the correct nodes have the same value after each step. If not, the agent receives a reward, otherwise she receives no reward and the game is over. Note that the King algorithm would technically not terminate but run for the full $f+1$ phases although all the correct nodes have the same value. But all-same validity tells us that the agent has no influence anymore once all correct nodes have the same value. For the evaluation we say the agent *wins* the game if the algorithm needs to run maximally long i.e. until the correct node is king.

5.2 Ben-Or Algorithm

The Ben-Or algorithm does not have distinct rounds, every phase directly corresponds to a step in the game. Important to note is, that the correct nodes only await $n-f$ messages, which means they can consist of anything between only those of correct nodes or those of $n-2f$ correct and f byzantine nodes. Effectively, the byzantine agent observes $n-f$ messages from all correct nodes and can then decide which values (at most f) she wants to replace with a value of her choice. The correct nodes then receive the modified $n-f$ messages. In general the agent can send different modifications to each correct node. The Ben-Or algorithm relies on the chance that all correct nodes get the same value from the coin, but there are three interesting variations which we will introduce below.

5.2.1 Fair Coin

Here we study the Ben-Or algorithm given in Section 3.3 without modifications. The byzantine agent observes the (binary) messages sent by the correct nodes. In contrast to the case of the King algorithm we do not need to give any additional information to the agent. The byzantine agent can only send the same modified message-vector to all correct nodes. This implies that her goal must be to let the correct nodes flip a coin, otherwise they all will be decided in the next two steps. After each step the agent receives a reward if she lets the correct nodes flip a coin. The game is over when a correct node has decided (Line 7 of Algorithm 2). It can happen that $\lfloor n/2 \rfloor + 2f + 1$ correct nodes flip the same value by chance, in this case the agent has won the game and receives an additional reward.

5.2.2 Predefined Coin

Here we consider a modification of Line 11 of Algorithm 2, where the coin is not random but predefined for each round to either be $Pr[x_u = 0] = 1$ or $Pr[x_u = 1] = 1$ (however, the sequence of the predefined coins is chosen randomly). The

byzantine agent now observes the coins value of the current and the next round in addition to the messages sent by the correct nodes. In this case it is obviously not optimal for the agent to let all correct nodes "flip" the coin since all will receive the same value. The agent needs to be allowed to send different modifications of the messages to different correct nodes.

Initial Value

Assuming the first value of the coin is $x_c = 1$ the byzantine nodes need to have initial values with a majority of 0's. Actually there need to be between $\lfloor n/2 \rfloor + 1$ and $\lfloor n/2 \rfloor + 2f$ zeros, such that the agent can send some correct nodes to "flip" and some to enter Line 9 of Algorithm 2. Whether she sends the majority to "flip" the coin or not, depends on the value of the next coin.

Termination

The algorithm terminates when all correct nodes have the same value. It is important to note, that the Ben Or algorithm does not need to terminate in this setting. However, this does not lead to problems during training as the agent always explores with a certain probability and therefore makes "mistakes", which lead to termination of the game.

5.2.3 Predefined Coin and Agent Selects Initial Values

This version is largely the same as the above, with the difference that the byzantine agent selects the initial values of the correct nodes. She observes a vector of 0's in the first step and decides for each entry, whether to leave it be a 0 or change it to a 1. We need to give the agent the additional information about whether it is the first step or not in her observation.

Chapter 6

Training and Results

For training we mostly use the default values for the parameters of the Stable Baseline's DQN Algorithm [20]. We increase the *exploration fraction* form 0.1 (default) to 0.2, leading to a longer period of exploration. The total length of the training period differs for the various versions. To monitor and evaluate the performance of the learner (agent) we focus on the following:

- Learning Curve
- Deterministic Validation Performance

The *learning curve* shows the (smoothed) rewards the agent achieved in each episode over the course of the training. Of course, we want the agent to learn, i.e. we want the rewards to increase the further we are in the learning process. On the other hand we want a stable performance of the agent, meaning we don't want to see large fluctuations in rewards between episodes.

By *deterministic validation performance* we mean, that we let the agent play **after** the training and measure her performance then. This is important, since the agent always explores during the training. Recall that the *exploration rate* is annealed to a final value of 0.02, meaning she takes 2% of her actions randomly. When we watch the agent play after the training the weights of the Q-network are kept fixed and she does not explore anymore. It is also important to keep the exploration in mind when studying the learning curves.

6.1 King Algorithm

6.1.1 One Byzantine Node

In this setting we train the agent to control only one byzantine node and use three correct nodes. Algorithm 1 tells us, we need at least 4 nodes in total in this case. The action space is small, the agent can only send 0, 1, or nothing. We set the training period to 200,000 steps.

Learning Curve

The learning curve (Figure 6.1a) shows, that the agent has learned and reaches the maximal reward of 5 frequently after the exploration fraction.

Deterministic Validation Performance

During the test the byzantine agent wins the game 1000/1000 times. Her strategy consists of making sure the correct nodes can't propose anything (Line 5 of Algorithm 1), to this end she broadcasts nothing if there is a majority of 0's in the correct nodes' values, and 0 if there is a majority of 1's. Then she herself can propose anything, she chose to always propose 0. Note that no correct node enters Line 8, because there is only 1 proposal. When the agent is king, she proposes nothing, meaning no correct node changes its value.

6.1.2 Five Byzantine Nodes

In this setting we train the agent to control five byzantine nodes and use 11 correct nodes, fulfilling the condition of $f < n/3$. The training period has a length of 3,000,000 steps.

Learning Curve

The learning curve looks promising (see Figure 6.1b). It shows that the agent reaches the maximum reward of 17 frequently.

Deterministic Validation Performance

Again, we evaluate the agent's performance after training, letting her play 1000 games. Out of those, she wins 991. Her strategy is similar to before, with the first action of each phase she makes sure none of the correct nodes propose anything, leaving her to be able to propose anything. Again, all byzantine kings propose nothing.

The nine cases where the agent lost, all followed the same pattern. The correct nodes all have starting value 0, except one correct node starts with 1. Note that it is still possible for the agent to win in this case, but the agent has little room for error. The agent then makes the mistake in the first step of the last phase, where she sends 0 for at least one of the byzantine nodes, letting the correct nodes reach agreement on 0. We argue this happens because the last phase was rarely explored during the training, especially during the exploration fraction. It is important to keep in mind here, that the number of the phase is part of

the observation, since the agent has therefore rarely observed the described case during training.

6.2 Ben-Or Algorithm

6.2.1 Fair Coin

We train the agent to control only one byzantine node and use 10 correct nodes, since Algorithm 2 requires $f < n/10$. We use 1,000,000 steps for training.

Learning Curve

The learning curve (Figure 6.2a) shows a continued learning process of the agent over the course of the training. There is no global maximum reward anymore, since the termination of the game is partly stochastic, due to the coin.

Deterministic Validation Performance

We consider the game to be won, when the algorithm terminated because $\lfloor n/2 \rfloor + 2f + 1$ correct nodes received the same value from the coin. Thereafter the agent has no influence anymore, since she can only send the same message to all correct nodes. We let the trained agent play 1000 games and she wins all of them. The agent's actions are simple, she works towards an equal number of 0's and 1's in the messages she sends to the correct nodes. If she allowed a majority of seven identical values, no correct node would flip a coin and the algorithm would terminate within the next two steps (see Line 7, Algorithm 2).

6.2.2 Predefined Coin

We train the agent for 2,000,000 steps, where we again have one byzantine node and 10 correct nodes.

Learning Curve

The learning curve shows a quick and steep learning process (Figure 6.2b). However, the rewards fluctuate a lot after the exploration fraction. As pointed out in Section 5.2.2, the algorithm does not need to terminate at all. If it does it is due to either exploration or a mistake by the agent. It is therefore even more crucial to focus on the deterministic validation performance.

Deterministic Validation Performance

Because the algorithm can run forever, given the agent has learned, we impose a maximum number of 10,000 phases after which we stop the game and consider the agent to have won. The trained agent wins 1000/1000 games. Essentially the agent's actions follow the principle by which we set the initial value, described in Section 5.2.2. The agent sends part of the correct nodes to the coin, how many depends on two aspects:

- The agent can not allow for any majority of more than $\lfloor n/2 \rfloor + 2f$ to form, since she would then have no influence in the next round.
- The value of the next coin: If the next coin will have the same value as the current coin, the agent wants the majority not to flip the coin. If the next coin will have the opposite value from the current, the agent lets the majority flip.

6.2.3 Predefined Coin and Agent Selects Initial Values

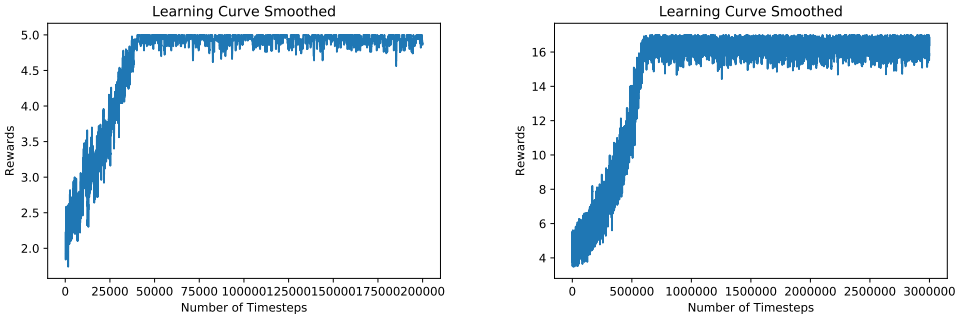
The agent's training consists of 5,000,000 steps in the setting of 10 correct nodes and one byzantine node.

Learning Curve

The learning curve (Figure 6.2c) looks similar to the previous case. Again, we observe large fluctuations in the reward, but want to emphasize the importance of deterministic validation.

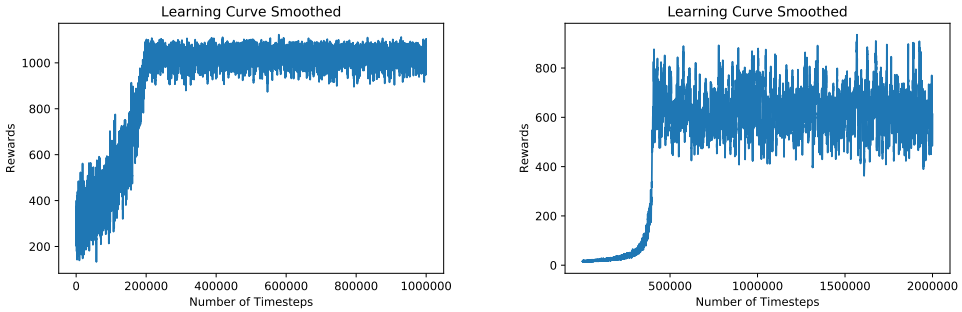
Deterministic Validation Performance

We again set a maximum number of phases of 10,000 and consider the agent to have won thereafter. Our trained agent wins 1000/1000 games. It is interesting for us to study the agent's behaviour in the first step of a game, during the rest she develops a similar strategy as in the previous case. For the first step she selects a vector of initial values, where 4 correct nodes have the value of the first coin, hence creating a majority for the opposite value to that of the first coin. This is consistent with what we described in Section 5.2.2.

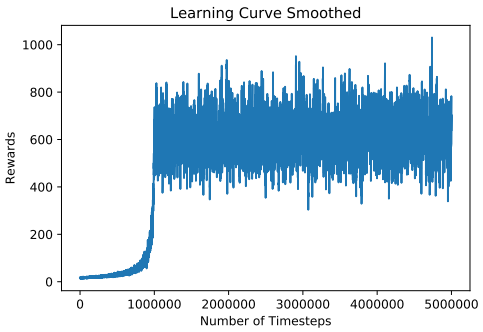


(a) 3 correct nodes, 1 byzantine node (b) 11 correct nodes, 5 byzantine node

Figure 6.1: King algorithm: learning curves smoothed by a moving average of 50 episodes, using code from [22]



(a) Fair coin (b) Predefined coin



(c) Predefined coin, agent selects initial values

Figure 6.2: Ben-Or algorithm: learning curves smoothed by a moving average of 50 episodes, using code from [22]

Chapter 7

Discussion

Our results show, that Deep Reinforcement Learning can be effectively used to learn and simulate (worst-case) byzantine behaviour. The agents develop optimal strategies for almost all settings, in terms of achieving maximal run time of each algorithm. Furthermore we can study the agents' actions, allowing us to potentially gain insights into approaches we did not think of. In some cases, the agents' actions are precisely in line with our own understanding, most obviously when the byzantine agent chooses the starting conditions for the correct nodes in Section 6.2.3.

We were able to formulate our problem as a Deep Reinforcement Learning task and to encode the full algorithm in the environment. Although the reward function is not unique, it is straightforward to reward the agent for every step where the algorithm does not terminate, in order to achieve maximal run time. Therefore, the problem is of a sensible form to apply Deep Reinforcement Learning. We want to highlight again, that the agent was never given other information about the algorithms than the observations and the rewards. Especially, did we not pre-train by feeding the agent certain actions, i.e. we did not show her strategies we considered useful.

Studying the results for the King Algorithm with five byzantine nodes, we found that the agent can run into trouble if a certain observation only occurs at a late stage of the game, which might not be reached often during exploration. In our case, we always used the same sequence of byzantine kings. Especially, in the last phase the king was always a correct node. Working with permutations of that sequence might improve the learning results. However, this is non-trivial, because a correct node being king earlier reduces the maximum run time of the algorithm.

Given the success of our experiments, we would be interested to see Deep Reinforcement Learning to be applied to other algorithms for the synchronous or asynchronous models in Byzantine Agreement. An interesting extension would be to use *self-play*, where we train not only the byzantine nodes but also the response of the correct nodes. Thereby effectively learning a new algorithm to solve the Byzantine Agreement problem.

Bibliography

[1] R. Wattenhofer, "Byzantine agreement," Oct. 2018, (secondary source). [Online]. Available: https://disco.ethz.ch/courses/hs18/distsys/lnotes/chapter11.pdf

[2] P. Berman, J. A. Garay, and K. J. Perry, "Towards optimal distributed consensus (extended abstract)," in *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, 1989, pp. 410–415.

[3] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols." in *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM, 1983, pp. 27–30.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[5] K. Nakashima, S. Kamiya, K. Ohtsu, K. Yamamoto, T. Nishio, and M. Morikura, "Deep reinforcement learning-based channel allocation for wireless lans with graph convolutional networks," 2019.

[6] P. Almasan, J. Suárez-Varela, A. Badia-Sampera, K. Rusek, P. Barlet-Ros, and A. Cabellos-Aparicio, "Deep reinforcement learning meets graph neural networks: exploring a routing optimization use case," 2019.

[7] C. Hua, S. X. Ding, and Y. A. Shardt, "A new method for fault tolerant control through q-learning," *IFAC-PapersOnLine*, vol. 51, no. 24, pp. 38 – 45, 2018, 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes SAFEPROCESS 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405896318322171

[8] F. B. Mismar and B. L. Evans, "Deep q-learning for self-organizing networks fault management and radio performance improvement," *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, Oct 2018. [Online]. Available: http://dx.doi.org/10.1109/ACSSC.2018.8645083

[9] J. Lee, J. Won, and J. Lee, "Crowd simulation by deep reinforcement learning," in *Proceedings of Motion, Interaction and Gamesm Limassol, Cyprus, November 8-10*, 2018.

[10] C. Hou, M. Zhou, Y. Ji, P. Daian, F. Tramer, G. Fanti, and A. Juels, "Squirrl: Automating attack discovery on blockchain incentive mechanisms with deep reinforcement learning," 2019.

[11] Y. Chen, L. Su, and J. Xu, "Distributed statistical machine learning in adversarial settings: Byzantine gradient descent," 2017.

[12] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 119–129. [Online]. Available: http://papers.nips.cc/paper/6617-machine-learning-with-adversaries-byzantine-tolerant-gradient-descent.pdf

[13] E. M. E. Mhamdi, R. Guerraoui, and S. Rouault, "The hidden vulnerability of distributed learning in byzantium," 2018.

[14] D. Yin, Y. Chen, K. Ramchandran, and P. Bartlett, "Defending against saddle point attack in byzantine-robust distributed learning," 2018.

[15] R. Wattenhofer, "Consensus," Oct. 2018, (secondary source). [Online]. Available: https://disco.ethz.ch/courses/hs18/distsys/lnotes/chapter8.pdf

[16] M. C. Pease, R. E. Shostak, and L. Lamport, "Reach- ing agreement in the presence of faults." *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[18] P. D. Christopher Watkins, "Technical note: Q-learning," *Machine Learning*, vol. 8, pp. 279–292, 05 1992.

[19] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *CoRR*, vol. abs/1708.05866, 2017. [Online]. Available: http://arxiv.org/abs/1708.05866

[20] OpenAI, "stable-baselines," Dec. 2019. [Online]. Available: https://github.com/hill-a/stable-baselines

[21] OpenAI, "Gym," Dec. 2019. [Online]. Available: https://github.com/openai/gym

[22] Stable-Baselines, Dec. 2019. [Online]. Available: https://colab.research.google.com/drive/1L_IMo6v0a0ALK8nefZm6PqPSy0vZIWBT#scrollTo=mPXYbV39DiCj

Appendix A

Proofs

A.1 Proof of Theorem 3.3

Assume an algorithm reaches byzantine agreement for $f \geq \lceil n/3 \rceil$. Consider now three groups of size $n/3$, $\lceil n/3 \rceil$ or $\lfloor n/3 \rfloor$. Where the group of size $\lceil n/3 \rceil$ consists of all byzantine nodes and in one group (correct nodes) all have value 1, in the other group all have value 0. The byzantine group sends different values to each other group, always supporting the value of the group, such that every correct node observers its own value $n - f$ times. By all-same validity each correct node will decide on its own value and therefore agreement is violated. [16] [1]

A.2 Proof of Lemma 3.6

Case 1: All correct nodes change to the king's value, then all correct nodes have the same value. [1]

Case 2: Some nodes do not change to the king's value, implying they have received a value x_p proposed $n - f$ times. Therefore, all other correct nodes have received the same value x_p $n - 2f$ times and have set their value to x_p (Line 8 of Algorithm 1) including the correct king. [1]

Only one value can be proposed more than f times. In order to propose a value a node must have received it $n - f$ times (Line 4 of Algorithm 1) of which $n - 2f$ must have come from correct nodes. If two correct nodes propose different values, it suggests there are $2(n - 2f) + f = 2n - 3f \geq n$ nodes in the system (using $3f < n$), which is a contradiction. [1]